

Algorithmen und Datenstrukturen

January 2, 2023

Inhaltsverzeichnis

1 Mathematische Grundlagen

- 1.1 Reihen
- 1.2 Potenzen und Logarithmen
- 1.3 Notationskonventionen
- 1.4 Grundbegriffe der Graphentheorie

2 Rekursive Algorithmen

- 2.1 Prinzip der Rekursion
- 2.2 Korrektheit rekursiver Algorithmen
- 2.3 Rekursive Berechnung der Potenzmenge
- 2.4 Algorithmenprinzip "Backtracking"

3 Analyse von Algorithmen

- 3.1 Korrektheit
- 3.2 Komplexität von Algorithmen
- 3.3 Komplexitätsanalyse wohlstrukturierter Algorithmen

4 Sortierverfahren

- 4.1 Vergleichsbasierte Sortierverfahren
- 4.2 Heapsort
- 4.3 Rekursives Sortieren nach "Teile und Herrsche"
- 4.4 Quicksort
- 4.5 Mergesort
- 4.6 Eigenschaften der Sortierverfahren
- 4.7 Nicht-vergleichsbasierte Sortierverfahren

5 Einfache Datenstrukturen

- 5.1 Abstrakte und konkrete Datenstrukturen
- 5.2 Abstrakter Datentyp Stack
- 5.3 Konkreter Datentyp ArrayStack
- 5.4 Amortisierte Laufzeitanalyse
- 5.5 Abstrakter Datentyp Warteschlange
- 5.6 Konkreter Datentyp Ringpuffer

1 Mathematische Grundlagen

1.1 Reihen

Arithmetische Reihe

- **Allgemeine arithmetische Reihe:** $a_0 + (a_0 + d) + (a_0 + 2d) + \dots + (a_0 + n \cdot d)$

$$\sum_{i=0}^n (a_0 + i \cdot d) = (n+1) \left(a_0 + d \frac{n}{2} \right)$$

Beispiel: Summe der ungeraden Zahlen von 1 bis 99, d.h. $1 + 3 + 5 + \dots + 99$:

$$a_0 = 1$$

$$d = 2$$

$$n = 49$$

Ergebnis: $50 \cdot (1 + 2 \cdot 49/2) = 2500$

- **Gaußsche Summenformel:** $1 + 2 + 3 + \dots + n$, also Summe der natürlichen Zahlen von 1 bis n . Dies ist der Spezialfall mit $a_0 = 0$; $d = 1$.

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Beispiel: Summe der Zahlen von 1 bis 50:

$$50 \cdot 51 / 2 = 1275$$



wichtig

1.2 Potenzen und Logarithmen

Der Logarithmus ist die Inverse der Potenzfunktion. $\log_a(x) = y \iff a^y = x$

spezielle Logarithmen:

$\lg(x) = \log_2(x)$, $lg(x) = \log_{10}(x)$, $\ln(x) = \log_e(x)$

1.3 Notationskonventionen

$\lceil x \rceil$ zur nächsten ganzen Zahl aufrunden

$\lfloor x \rfloor$ zur nächsten ganzen Zahl abrunden

$[a..b] = x | a \leq x \wedge x \leq b$ mit Intervallgrenzen

$]a..b[= x | a < x \wedge x < b$ ohne Intervallgrenzen

$arr[i..k]$ Teilfolge der Elemente von $arr[i]$ bis $arr[k]$

1.4 Grundbegriffe der Graphentheorie

Graphen bestehen aus einer Menge von Knoten und Kanten, die diese verbinden.

Ein Graph ist gerichtet, wenn die Kanten eine Richtung haben.

Für einen Knoten v eines gerichteten Graphen $G = (V, E)$ ist der Eingangsgrad $\text{indeg}(v)$ die Anzahl der Kanten, die in v enden, und der Ausgangsgrad $\text{outdeg}(v)$ die Anzahl der Kanten, die von v ausgehen.

Ein Zyklus ist ein Weg der bei einem Knoten startet und endet.

Ein gerichteter Graph ist zusammenhängend, wenn es einen Weg zwischen jedem Knotenpaar gibt.

Ein Baum hat einen Knoten als Wurzel, jeder Knoten hat genau einen Vorgänger und ist zusammenhängend.

Ein Knoten ohne Kinder heißt Blatt. Ein leerer Baum hat die Höhe 0. Ein Binärbaum ist ein Baum, dessen Knoten maximal zwei Kinder haben.

Traversierungen: Preorder (WLR), Inorder (LWR), Postorder (LRW)

2 Rekursive Algorithmen

2.1 Prinzip der Rekursion

Ein rekursiver Algorithmus besteht aus einem Basisfall und einem rekursiven Aufruf. Der rekursive Aufruf muss immer kleiner werden, damit die Rekursion endet. Die Rekursion kann durch eine Schleife ersetzt werden.

```
public static double sum_v2(double[] arr) {
    return sum_v2(arr, 0, arr.length-1);
}
/** Berechnet Summe der Werte von arr[firstIndex..lastIndex] */
private static double sum_v2(double[] arr, int firstIndex, int lastIndex) {
    if (firstIndex == lastIndex) {
        // zu summierender Bereich besteht nur aus einem Element
        return arr[firstIndex];
    }
    else {
        int mid = (firstIndex + lastIndex) / 2;
        return sum_v2(arr, firstIndex, mid) + sum_v2(arr, mid+1, lastIndex);
    }
}
```

2.2 Korrektheit rekursiver Algorithmen

Ein Beweisverfahren ist die Berechnungsinduktion.

Beweis mittels Berechnungsinduktion. Eigenschaft, die nachgewiesen werden soll:

$$P((x,n),y) :\Leftrightarrow x^n = y$$

d.h. wir wollen zeigen, dass $h(x,n) = x^n$ gilt.

- **Induktionsbasis** (Argumente führen zu keinem rekursiven Aufruf)

Fall $n = 0$:

Ergebnis: $h(x, n) = 1 = x^0$, d.h. Eigenschaft erfüllt

- **Induktionsschritte** (Argumente führen zu rekursiven Aufrufen):

Wir können als Induktionshypothese für die rekursiven Aufrufe verwenden, dass $h(z_i, k_i) = z_i^{k_i}$ gilt.

Fall $n > 0$, n gerade:

Ergebnis laut Programmcode: $h(x,n) = y*y$, wobei $y = h(x, n/2)$

Wir verwenden die Induktionshypothese: es gilt $h(x, n/2) = x^{n/2}$

somit $h(x,n) = y*y = x^{n/2} * x^{n/2} = x^n$,

d.h. Eigenschaft in diesem Fall auch für n erfüllt.

Fall $n > 0$, n ungerade:

Ergebnis laut Programmcode: $h(x,n) = x * h(x,n-1)$

Wir verwenden die Induktionshypothese: es gilt $h(x, n-1) = x^{n-1}$

Somit $h(x,n) = x * h(x,n-1) = x * x^{n-1} = x^n$,

d.h. Eigenschaft in diesem Fall auch für n erfüllt

- **Induktionsschluss:** somit folgt für alle $n \geq 0$, dass $h(x,n) = x^n$ gilt.

2.3 Rekursive Berechnung der Potenzmenge

Beispiel:

Menge: $M = \{a, b, c\}$

Potenzmenge: $\rho(M) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

2.3.1 Rekursiver Lösungsansatz

- in welchen einfachen Fällen kann die Lösung direkt angegeben werden?
der einfachste Fall ist die leere Menge $M = \emptyset$
die leere Menge hat nur sich als Teilmenge $\rho(\emptyset) = \{\emptyset\}$
- Wie können in nicht einfachen Fällen die Teilmengen bestimmt werden?
Sei Menge $M = \{a_1, \dots, a_{n-1}, a_n\}$ nicht leer ($n \geq 1$)

1. Wir wählen ein Element der Menge, z.B. a_n

2. Es gibt nun zwei Arten von Teilmengen:

T^+ Teilmengen, die das Element a_n enthalten

T^- Teilmengen, die das Element a_n nicht enthalten

Die Menge aller Teilmengen ist die Vereinigung von T^+ und T^- , d.h. $\rho(M) = T^+ \cup T^-$

Die Menge T^+ kann nun rekursiv berechnet werden, indem wir a_n aus M entfernen und die Potenzmenge von M berechnen.

Die Menge T^- ist die Potenzmenge von M ohne a_n .

Die Potenzmenge von M ist also die Vereinigung von T^+ und T^- .

Beispiel: Wenn $M = \{a, b, c\}$

Wähle z.B. c als Element:

T^- : alle Teilmengen ohne c , also alle Teilmengen von $\{a, b\}$

$T^- = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

T^+ : alle Teilmengen mit c , Nimm zu jeder Teilmenge von T^- und füge c hinzu

$T^+ = \{\{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

Insgesamt: $\rho(\{M\}) = T^+ \cup T^- = \{\emptyset, \{a\}, \{b\}, \{a, b\}, \{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

2.3.2 Algorithmischer Ansatz

Falls M leer ($M = \emptyset$)

leere Menge ist die einzige Teilmenge

Falls M nicht leer Wähle ein Element a_n aus M

Berechne Sammlung T^- aller Teilmengen von M ohne a_n (rekursiv)

Berechne Sammlung T^+ aller Teilmengen, die a_n enthalten:

Nimm dazu jede Menge aus T^- und bilde eine neue Menge, indem a_n hinzugefügt wird

Die Menge aller Teilmengen ist die Vereinigung von T^+ und T^-

```
private static <E> Set<Set<E>> allSubsets(E[] arr, int maxIndex) {
    Set<Set<E>> resultSet = new HashSet<Set<E>>();
    if (maxIndex >= 0) {
        // Menge ist nicht leer, waehle letztes Element im gegebenen Bereich
        E selected = arr[maxIndex];
        // Bilde rekursive alle Teilmengen ohne selected
        Set<Set<E>> resultSet1 = allSubsets(arr, maxIndex - 1);
        // nimm jede dieser Mengen zum Ergebnis hinzu
        resultSet.addAll(resultSet1);
        // bilde alle Teilmengen, die selected enthalten
        for (Set<E> set1 : resultSet1) {
            // Erzeuge Kopie der Menge aus resultSet1 und nimm gewaehltes Element dazu
```

```
Set<E> set2 = new HashSet<E>(set1);
set2.add(selected);
    // fuege die ergaenzte Kopie zum Ergebnis hinzu
resultSet.add(set2);
}
} else {
    // Menge ist leer. Leere Menge hat nur leere Menge als einzige Teilmenge
Set<E> emptySet = new HashSet<>();
resultSet.add(emptySet);
}
return resultSet;
}
```

2.4 Algorithmenprinzip "Backtracking"

2.4.1 Grundidee "Trial and Error"

1. Versuche eine Lösung zu finden
2. Wenn die Lösung nicht passt, versuche eine andere Lösung
3. Wenn keine Lösung passt, gehe zurück und versuche eine andere Lösung

3 Analyse von Algorithmen

3.1 Korrektheit

3.1.1 Insertionsort

- Am Anfang des Arrays wird ein sortierter Bereich aufgebaut, in den nach und nach die folgenden Elemente eingefügt werden (deshalb "Insertionsort").
- Am Anfang besteht der sortierte Bereich nur aus dem ersten Element $arr[0]$.
- In jedem Schritt wird ein Element $arr[i]$ aus dem unsortierten Bereich in den sortierten Bereich eingefügt.
- Wenn alle Elemente eingefügt wurden, ist das Array sortiert.



Definition:

Enthält ein Array arr am Anfang die n Elemente $arr[0], arr[1], \dots, arr[n-1]$, so ist das Array sortiert, wenn gilt:

- **Permutationen:** Die Elemente von arr sind eine Permutation (Umordnung) der ursprünglichen Elemente von $[0, n-1]$.
- **Monotonie:** Die Elemente von arr sind monoton steigend/fallend sortiert.

Mit **sortiert** ist sofern nicht anders angegeben immer **aufsteigend sortiert** gemeint.

3.2 Komplexität von Algorithmen

Komplexität bezeichnet den Ressourcenverbrauch von Algorithmen. Ressourcen sind dabei die Ausführungszeit und der Speicherbedarf.

Der Ressourcenbedarf hängt von mehreren Faktoren ab:

- Umfang der Daten (Größe des Problems)
- Zusammensetzung der Daten (aktuelle Sortierung der Daten)
- Ausführungsgeschwindigkeit und Speicherbedarf bei der Ausführung

3.2.1 Vorgehen bei der Laufzeitanalyse

- Bestimme für jede Anweisung A_j des Programms die Häufigkeit k_j der Ausführung.
- Die Gesamtkosten bei Problemgröße n können dann so zusammengezählt werden: $T(n) = \sum_{A_j}^n k_j \cdot c_j$
wobei k_j die Häufigkeit für die Ausführung von Anweisung A_j ist
und c_j die Einzel-Ausführungszeit für A_j ist.

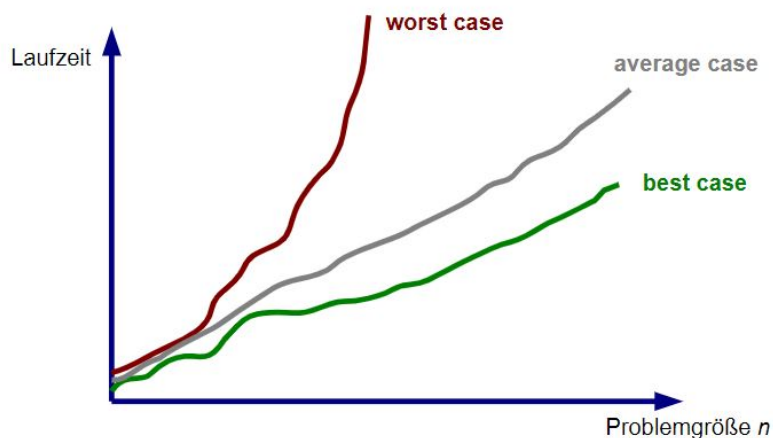
3.2.2 Abhängigkeit von der Datenzusammensetzung

Best Case: basiert auf Daten, die im Algorithmus eine minimale Anzahl von Schritten erfordern.

Worst Case: basiert auf Daten, die im Algorithmus eine maximale Anzahl von Schritten erfordern.

Average Case: basiert auf Daten, die im Algorithmus eine durchschnittliche Anzahl von Schritten erfordern.

Laufzeit in Abhängigkeit von der Problemgröße



3.2.3 Häufig verwendete Größenordnungen

- $O(1)$: Konstante Laufzeit - elementare Operationen
- $O(\log n)$: Logarithmische Laufzeit - binäre Suche
- $O(n)$: Lineare Laufzeit - lineare Suche
- $O(n \log n)$: Logarithmische Laufzeit - schnelle Sortieralgorithmen
- $O(n^2)$: Quadratische Laufzeit - einfache Sortieralgorithmen
- $O(n^3)$: Kubische Laufzeit
- $O(c^n)$: Exponentielle Laufzeit
- $O(n!)$: Permutationen berechnen

3.3 Komplexitätsanalyse wohlstrukturierter Algorithmen

Unter wohlstrukturierten Algorithmen versteht man solche, die nur mit Hilfe von Sequenz (nacheinander ausführen), Alternative (Fallunterscheidung) und Iteration (Schleife) definiert sind.

3.3.1 Lineare Suche

Bei der linearen Suche wird ein Array von links nach rechts durchsucht, bis das gesuchte Element gefunden wurde.

```
public String sucheNummer(String suchname) {  
    for (int i = 0; i < anzahl; i++) {  
        if (liste[i].name.equals(suchname))  
            return liste[i].nummer;} // gefunden  
return null;} // nicht gefunden
```

3.3.2 Binäre Suche

Bei sortierten Daten kann der Bereich, in dem der gesuchte Schlüssel liegen kann, nach und nach immer wieder halbiert werden. Dieses Verfahren wird **binäre Suche** genannt.

```
private String searchBinRek(String sname, int from, int to){  
    if (to < from) {  
        return null; //leerer Suchbereich, nichts gefunden  
    } else {  
        // Mitte des Suchbereichs berechnen  
        int middle = (from + to) / 2;  
        // Element in der Mitte vergleichen  
        int res = suchname.compareTo(liste[middle].name);  
        if (res < 0) {  
            // in unterer Haelfte weitersuchen  
            return searchBinRek(sname, from, middle-1);  
        } else if (res > 0) {  
            // in oberer Haelfte weitersuchen  
            return searchBinRek(sname, middle+1, to);  
        } else // (res == 0) {  
            // Schluessel gefunden  
            return liste[middle].nummer;  
        }  
    }  
}}
```

4 Sortiervverfahren

4.1 Vergleichsbasierte Sortiervverfahren

Interne und Externe Sortiervverfahren

- **Interne Sortiervverfahren:** alle Datensätze passen in den Hauptspeicher.
- **Externe Sortiervverfahren:** nur ein Teil der Datensätze passt in den Hauptspeicher.

Eine Ordnungsrelation heißt **total** wenn sie

- die Eigenschaften einer Halbordnung erfüllt (reflexiv, transitiv, antisymmetrisch)
 - reflexiv: $a \leq a$
 - transitiv: $a \leq b$ und $b \leq c \Rightarrow a \leq c$
 - antisymmetrisch: $a \leq b$ und $b \leq a \Rightarrow a = b$
- und alle Werte miteinander vergleichbar sind.

Vergleiche über Interface Comparable

```
public interface Comparable {  
    int compareTo(Object o) throws ClassCastException;  
}
```

wenn $x < y$ dann $x.compareTo(y) < 0$

wenn $x = y$ dann $x.compareTo(y) = 0$

wenn $x > y$ dann $x.compareTo(y) > 0$

4.1.1 Laufzeitkomplexität von Selection-Sort



Worst Case = Best Case = Average Case: $O(n^2)$

4.1.2 Laufzeitkomplexität von Bubble-Sort

Best Case: $O(n)$ - Daten sind schon aufsteigend sortiert

Worst Case: $O(n^2)$ - Daten sind absteigend sortiert

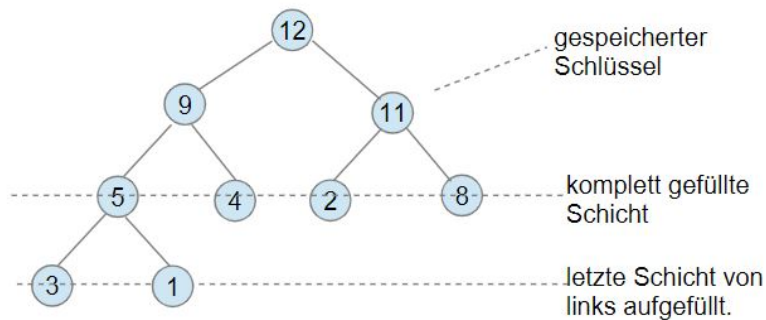
Average Case: $O(n^2)$

4.2 Heapsort

4.2.1 Binäre Maximum-Heaps

Ein **Maximum-Heap** ist ein binärer Baum, der folgende Eigenschaften erfüllt:

- Der Baum ist fast vollständig - alle Ebenen bis auf die letzte sind vollständig.
- Für jeden Knoten gilt, dass der Schlüssel des Knotens größer oder gleich dem Schlüssel der Kinder ist.



Speicherung eines fast vollständigen Baumes in einem Array:

Die Wurzel wird an Index 0 abgelegt

Hat ein Knoten den Index i , so sind die Kinder an den Indizes $2i + 1$ und $2i + 2$ abgelegt.

4.2.2 Sortieren mittels Heap

Grundprinzip von Heapsort:

1. Heap aufbauen: Array wird in einen Maximum-Heap umgewandelt.
2. Heap abbauen, sortierten Bereich aufbauen:
 - Vertausche Wurzel $a[0]$ und $a[k]$, Maximum ist nun am Ende des Arrays.
 - Heap-Eigenschaft wiederherstellen: Wurzel $a[0]$ nach unten schieben, bis Heap-Eigenschaft wiederhergestellt ist.

4.2.3 Laufzeitkomplexität von Heapsort

”versenken” (sink): $O(\log n)$

”erstelleMaxHeap” (createMaxHeap): $O(n \log n)$

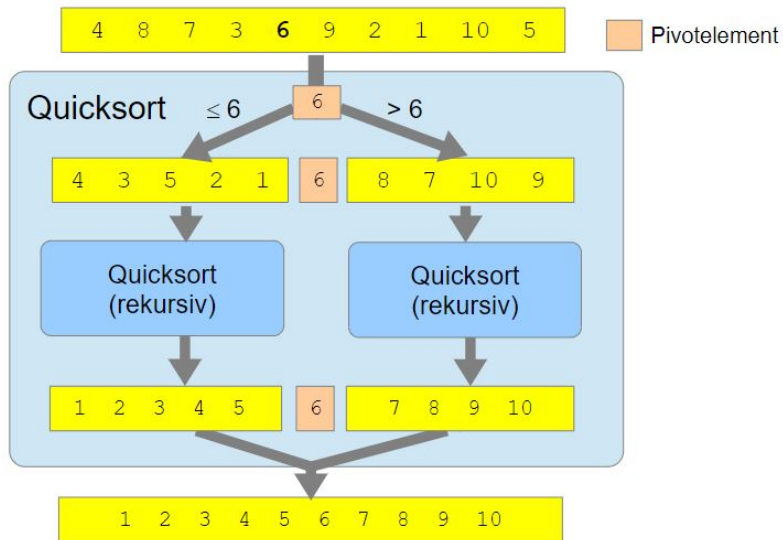
Heapsort: $O(n \log n)$

4.3 Rekursives Sortieren nach ”Teile und Herrsche”

1. **Divide**: Besteht die Datenmenge aus mehr als einem Element, so wird sie in zwei Teilmengen zerlegt.
2. **Conquer**: Die Teilmengen werden rekursiv sortiert.
3. **Combine**: Die Teilmengen werden zusammengeführt.

<i>divide</i>	<i>combine</i>	<i>Sortierverfahren</i>
Aufteilen in Werte, die größer bzw. kleiner als ein gewähltes Vergleichselement (sog. Pivotelement) sind	Sortierte Teilfolgen aneinander anhängen (bzw. stehen schon nacheinander im Feld), mit Pivotelement dazwischen.	\Rightarrow Quicksort
Aufteilen der zu sortierenden Werte in zwei gleich große Teilmengen (z.B. Arraybereich in der Mitte teilen).	Verschmelzen der sortierten Teilfolgen zu einer sortierten Gesamtfolge durch merge-Operation	\Rightarrow Mergesort

4.4 Quicksort



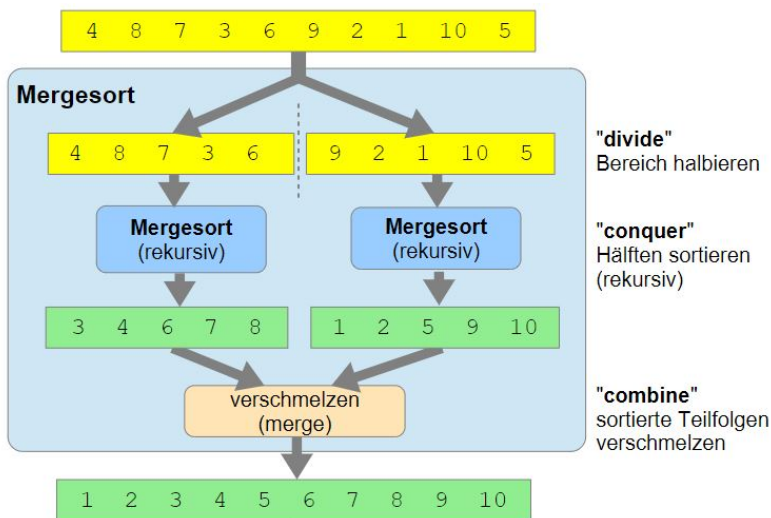
4.4.1 Laufzeitkomplexität von Quicksort

Worst Case: $O(n^2)$

Best Case: $O(n \log n)$

Average Case: $O(n \log n)$

4.5 Mergesort



```
void mergesort(double[] a, int von, int bis) {  
    if (bis - von > 0) {  
        //Mehr als ein Element zu sortieren  
        //Daten in zwei Haelften aufteilen  
        int mitte = (von + bis) / 2;  
        //linke und rechte Haelfte sortieren  
        mergesort(a, von, mitte);  
        mergesort(a, mitte+1, bis);  
        // Sortierte Teilfolgen a[von..mitte] und  
        // a[mitte+1..bis] miteinander verschmelzen  
        merge(a, von, mitte, bis);  
    }  
}
```

```
}
```

4.5.1 merge - Verschmelzen sortierter Teilfolgen

```
void merge(double[] a, int links, int mitte, int rechts) {
    //Kopie der linken Haelfte a[links..mitte] erzeugen
    double[] tmpLinks = new double[mitte - links + 1];
    for (int i = 0; i < tmpLinks.length; i++) {
        tmpLinks[i] = a[links + i];
    }
    // Inhalt von tmpLinks (linker Teil) und a[mitte+1 .. rechts] zu
    // sortierter Gesamtfolge im Ergebnisbereich a[links..rechts] verschmelzen
    int indexLinks = 0;
    int indexRechts = mitte+1;
    int indexErgebnis = links;
    while (indexLinks < tmpLinks.length && indexRechts <= rechts) {
        // linke und rechte Teilfolge enthalten noch Elemente
        // nimm kleineren Wert von beiden Teilfolgen als naechsten Wert
        if (tmpLinks[indexLinks] <= a[indexRechts]) {
            a[indexErgebnis] = tmpLinks[indexLinks];
            indexLinks++;
        } else {
            a[indexErgebnis] = a[indexRechts];
            indexRechts++;
        }
        indexErgebnis++;
    }
    // falls nur noch linke Teilfolge Werte enthaelt (rechte Teilfolge aufgebraucht),
    // uebertrage sie in das Ergebnis
    while (indexLinks < tmpLinks.length) {
        a[indexErgebnis] = tmpLinks[indexLinks];
        indexErgebnis++;
        indexLinks++;
    }
    // falls linke Teilfolge abgearbeitet ist und nur noch rechte Teilfolge Werte
    // enthaelt, stehen diese schon richtig im Ergebnisfeld a und muessen nicht
    // mehr behandelt werden
}
```

4.5.2 Laufzeitkomplexität von Mergesort

Worst Case = Best Case = Average Case: $O(n \log n)$

4.6 Eigenschaften der Sortiervverfahren

Stabile Sortiervverfahren: Ein Sortiervverfahren heißt stabil, wenn die relative Reihenfolge von Elementen, deren Schlüssel gleich sind, während des Sortiervorgangs beibehalten wird.

Bubble-Sort, Insertion-Sort, Mergesort sind stabile Sortiervverfahren.

Quicksort, Heapsort sind instabile Sortiervverfahren.

Selectionsort ist ein instabiles Sortiervverfahren, das aber stabil gemacht werden kann.

(für Arrays instabil -; für Listen stabil)

4.6.1 Zusammenfassung der Sortierverfahren

Verfahren	best case	average case	worst case	stabil	in-place-Verfahren
Bubblesort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	ja	ja
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	nein ¹⁾	ja
Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	ja	ja
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	nein	ja
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	ja	nein
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	nein	ja

4.7 Nicht-vergleichsbasierte Sortierverfahren

Jedes vergleichsbasierte Sortierverfahren benötigt zum Sortieren von n verschiedenen Schlüsseln im schlechtesten Fall (und auch im mittleren Fall) $O(n \log n)$ Schlüsselvergleiche.

4.7.1 Eigenschaften eines Entscheidungsbaums

- Ein Binärbaum repräsentiert die beim Suchen durchgeführten Vergleiche.
- Wurzelknoten repräsentiert den ersten Vergleich.
- Verzweigung nach links repräsentiert den Fall, dass der Vergleich wahr ist.
- Der Worst-Case ist der Pfad mit der größten Tiefe.

4.7.2 Bucket-Sort

Idee: Die Elemente werden in Buckets sortiert, die jeweils einen Bereich von Schlüsseln repräsentieren. Die Buckets werden dann einzeln sortiert und anschließend zusammengeführt.

gegeben:

- ▶ Schlüsselbereich $[0, 100[$
- ▶ $n = 12$ zu sortierende Werte: 78, 17, 39, 26, 72, 94, 21, 12, 23, 68, 77, 42
- ▶ vorgegebene Konstante $c = 3$

Sortieren mit Bucketsort:

- ▶ Es sind $m = n/c = 12/3 = 4$ Fächer zu bilden.
- ▶ Berechnung des Fachindex für Schlüssel k nach der Formel $F(k) = \lfloor k/25 \rfloor$
- ▶ Ablauf:

(1) Werte durchgehen und auf Fächer (buckets) verteilen:

Fachindex	Schlüsselbereich für Fach	Werteliste
0	$[0, 25[$	17, 21, 12, 23
1	$[25, 50[$	39, 26, 42
2	$[50, 75[$	72, 68,
3	$[75, 100[$	78, 94, 77

(2) Fächer sortieren mit üblichem Sortierverfahren (z.B. Insertionsort)

Fachindex	Schlüsselbereich für Fach	Werteliste
0	$[0, 25[$	12, 17, 21, 23
1	$[25, 50[$	26, 39, 42
2	$[50, 75[$	68, 72
3	$[75, 100[$	77, 78, 94

(3) Fächer aufsteigend durchgehen und Werte einsammeln.

Ergebnis:

12, 17, 21, 23, 26, 39, 42, 68, 72, 77, 78, 94

Laufzeitkomplexität:

Best Case = Average Case: $O(n)$

Worst Case $O(n^2)$

4.7.3 Radix-Sort

Wiederhole für jede Stelle p der insgesamt d Stellen, von der niederwertigsten Stelle rechts bis zur höchstwertigen Stelle links jeweils folgende Partitionierungs- und Sammelphase

(1) Partitionierungsphase: Gehe die Wertefolge durch und ordne sie entsprechend dem Zeichen an der Stelle p in das zugehörige Fach ein.

wichtig: neue Elemente werden hinten an die Fachliste angehängt!

(2) Sammelphase: Die Einträge aus den Fächern aufsammeln und wieder in eine lineare Folge bringen

- bei Fach mit niedrigstem Wert beginnen

- die im Fach vorgegebene Reihenfolge muss beibehalten werden

Diese Folge wird dann beim nächsten Durchgang für die Partitionierung verwendet.

Nach d Durchgängen durch alle Stellen sind die Daten richtig sortiert.

Laufzeitkomplexität:

Best Case = Worst Case = Average Case: $O(d \cdot n) = O(n)$

5 Einfache Datenstrukturen

5.1 Abstrakte und konkrete Datenstrukturen

Abstrakte Datentypen (ADT):

Ein ADT definiert eine Datenstruktur unabhängig von ihrer konkreten Implementierung.

Konkrete Datenstrukturen:

Ein konkreter Datentyp ist die Implementierung eines ADT.

5.2 Abstrakter Datentyp Stack

Stack: Eine Sammlung von Elementen, die nach dem LIFO-Prinzip (Last In First Out) verwaltet wird.

```
public interface Stack {  
    void push(Object elem); // Element oben auf dem Stack ablegen  
    Object pop() throws EmptyStackError;  
    // oberstes Element des Stacks entnehmen und zurueckgeben  
    boolean isEmpty(); // prueft, ob Stack leer ist  
}
```

5.3 Konkreter Datentyp ArrayStack

Bei Stacks kann man noch zwei Varianten unterscheiden:

ob sie eine begrenzte Kapazität haben oder sie unbeschränkt wachsen können.

Laufzeitkomplexität für ArrayStack mit begrenzter Kapazität: $O(1)$

5.4 Amortisierte Laufzeitanalyse

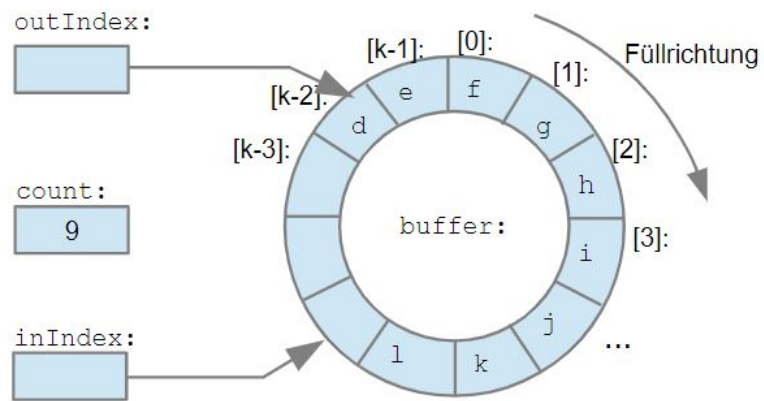
Die Laufzeit eines Algorithmus wird über mehrere Operationen hinweg betrachtet.

5.5 Abstrakter Datentyp Warteschlange

Eine Queue ist eine Sammlung von Elementen, die nach dem FIFO-Prinzip (First In First Out) verwaltet wird.

```
public interface Queue {  
    void enqueue(Object elem); // fuege Element vorne ein  
    Object dequeue() throws EmptyQueueError;  
    // entnimmt Element hinten  
    boolean isEmpty(); // pruefen, ob Warteschlange leer ist  
}
```

5.6 Konkreter Datentyp Ringpuffer



Laufzeitkomplexität: $O(1)$